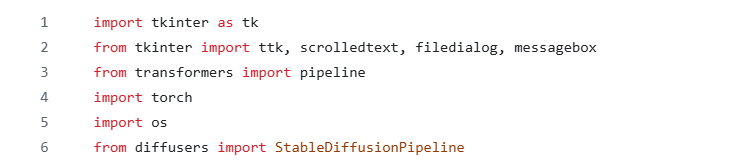
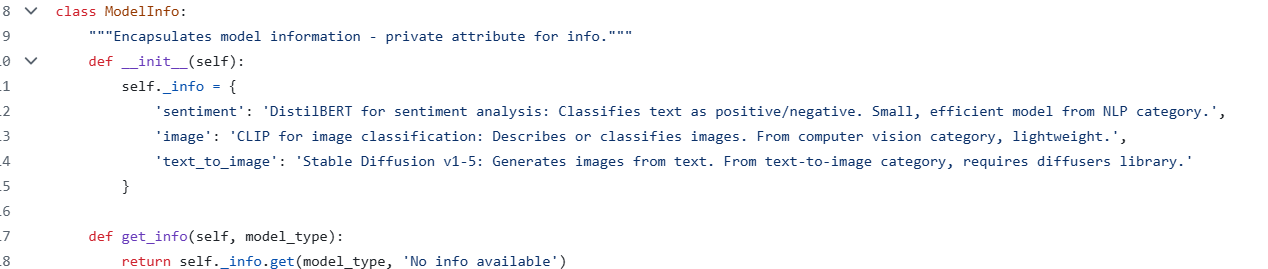
**Code Explanation:**

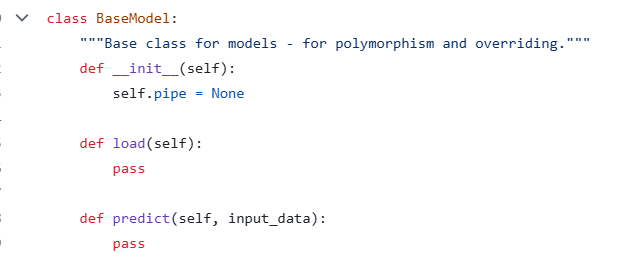
The Python program we demonstrated here is a desktop application that combines artificial intelligence models with a graphical user interface (GUI). The user interface was created using the tkinter library, and it uses well-known machine learning libraries like transformers and diffusers to execute tasks such as sentiment analysis, image categorization, and text-to-image generation. It is more than just a simple tool; it demonstrates object-oriented programming (OOP) concepts such as encapsulation, inheritance, polymorphism, multiple inheritance, and decorators. By arranging the program in this manner, the code not only delivers useful functionality but also acts as an instructional example of how software engineering principles may be used to organize and simplify complicated processes.



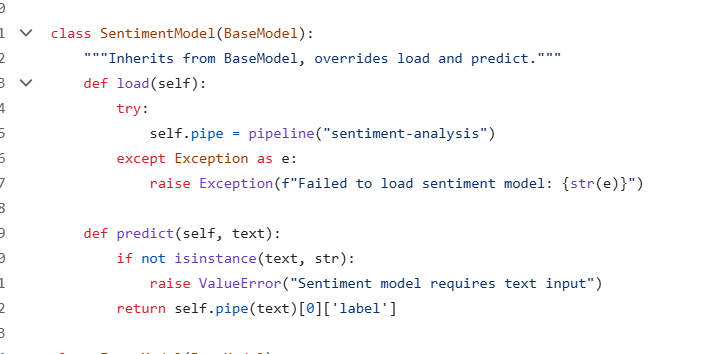
At first we have started his program by importing all required libraries. The tkinter family (tk, ttk, scrolledtext, filedialog, and messagebox) forms the GUI layer: it provides widgets, a scrollable text area, file pickers, and user-friendly dialogs for errors and messages. The AI functionality is delivered by Hugging Face’s transformers—exposed through the high-level pipeline interface—and by diffusers for text-to-image generation via StableDiffusionPipeline. Although torch (PyTorch) is not called explicitly, it underpins model computation. We used os for file-system checks such as verifying image paths. Together these imports make it possible to build an interactive desktop app that runs modern pretrained models.



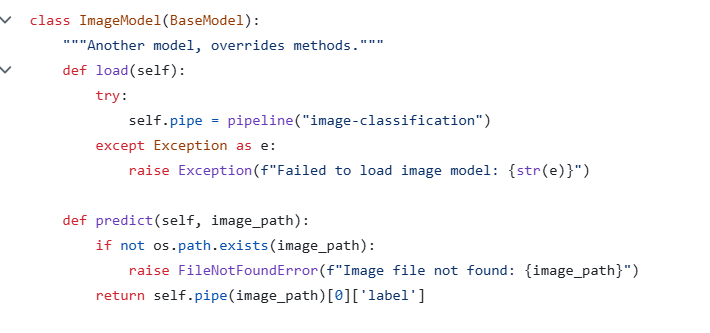
We used different classes in this program. The first class defined is ModelInfo, that encapsulates information about each model in the application. It holds a private dictionary of descriptive text about sentiment analysis, image classification, and text-to-image generation. By using a private attribute and exposing a public method called get\_info, the class demonstrates encapsulation, ensuring that internal data cannot be directly modified but can be safely accessed when needed.



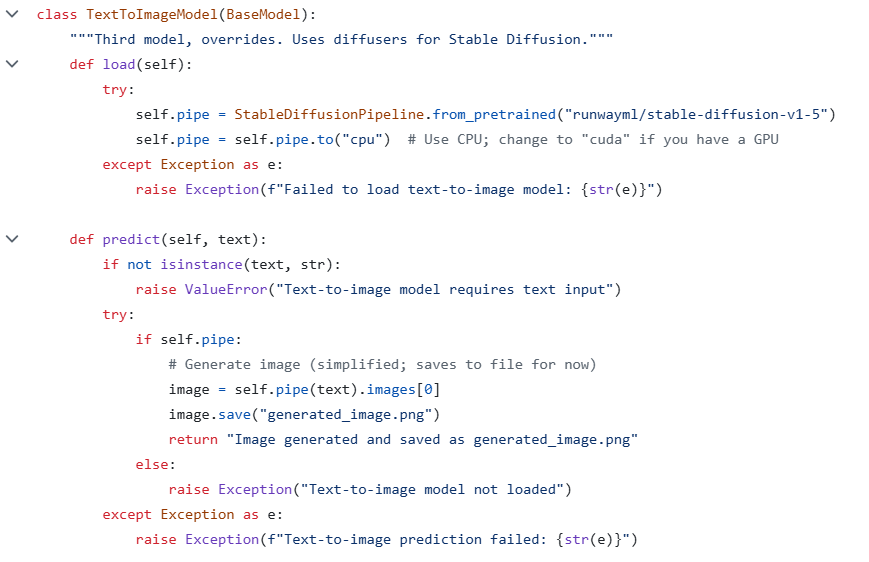
Secondly, we used another class named BaseModel that establishes a common interface and shared state for all concrete AI models. We used the attribute self.pipe later to hold the loaded inference pipeline, while the load and predict methods are intentionally left empty. Subclasses will override these methods to implement task-specific loading and inference. This pattern enables **inheritance** and prepares the ground for **polymorphism**, since the GUI can interact with any model through the same method names. The subclasses—SentimentModel, ImageModel, and TextToImageModel—each provide specific behavior.



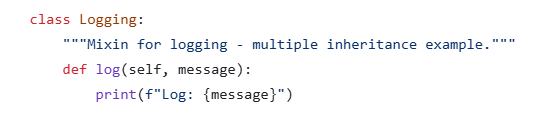
The SentimentModel class wraps Hugging Face’ pipeline("sentiment-analysis"), which usually pulls in a lightweight DistilBERT model fine-tuned for sentiment tasks. Its predict method first checks that the input is actually text, then runs the pipeline and returns the top label (like “POSITIVE” or “NEGATIVE”). If the model can’t be loaded, it raises a clear, user-friendly error instead of crashing—solid defensive programming for a GUI app.



We created the class ImageModel to provide image classification via pipeline("image-classification"). This class typically defaults to a Vision Transformer model. Before inference, the predict method checks that the supplied path points to an actual file, preventing cryptic errors later in the stack. The method returns the top predicted class label and mirrors the structure of SentimentModel, reinforcing a uniform interface across different modalities.



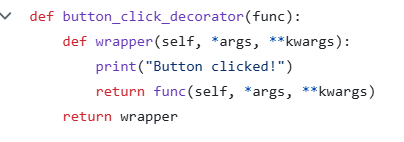
Another class TextToImageModel integrates Stable Diffusion v1-5 using diffusers. The load method instantiates the pipeline and targets the CPU device for portability, though a GPU (“cuda”) would substantially improve performance. The predict method expects a text prompt, generates an image, and saves it to generated\_image.png, returning a succinct status message. Structured exceptions provide clear feedback if loading fails or if inference encounters problems, which is especially important for long-running generative tasks.



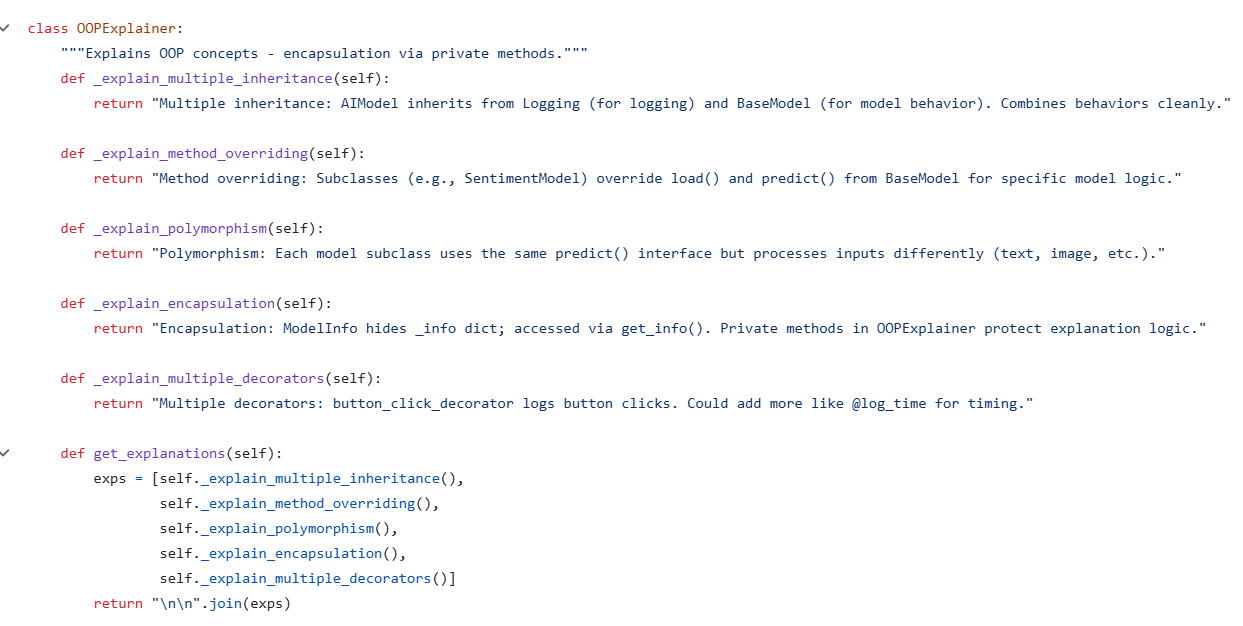
Logging is a mixin class that supplies a simple log method. We have designed it to combine with other classes through **multiple inheritance** to add cross-cutting behavior (in this case, console logging) without polluting the domain logic of the model classes themselves. This is a common Python idiom for lightweight aspect-oriented concerns.



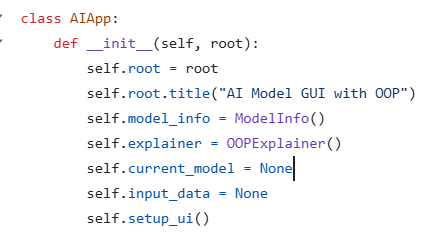
Now, the class AIModel combines Logging and BaseModel to act as a **wrapper and factory** for the concrete models. Its constructor receives a string key and delegates instance creation to \_create\_model, which returns the appropriate subclass. Both load and predict forward calls to the inner model\_instance, while Logging.log records milestone events such as a successful load. This design centralises model selection, keeps the GUI decoupled from concrete classes, and exhibits **polymorphism**: the GUI can call predict on AIModel without knowing whether it is operating on text, an image, or a generative model.



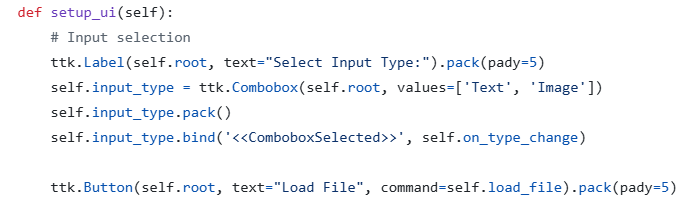
button\_click\_decorator is a higher-order function that wraps another function and injects a side effect—in this case, printing “Button clicked!”—before delegating to the original function. Applying this decorator to a GUI handler demonstrates how **decorators** can extend behavior orthogonally, without modifying the wrapped function’s internal logic.



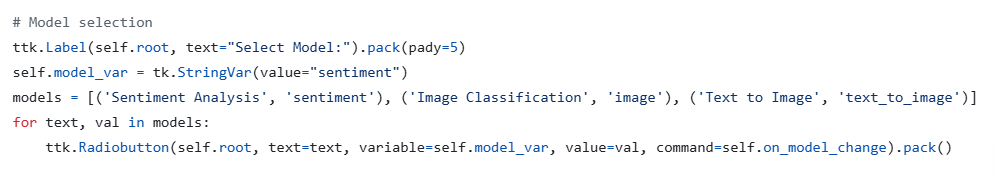
OOPExplainer provides self-contained textual descriptions of the OOP principles embodied in the code. Each explanation method is marked private by convention (single underscore) to indicate they are internal helpers. The get\_explanations method aggregates these texts into a single formatted string. Beyond their didactic value, these explanations become live content displayed in the GUI, linking theory to the concrete implementation the user is operating.



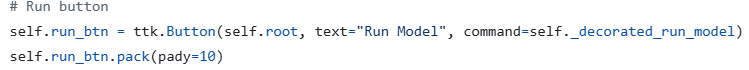
AIApp orchestrates the GUI. The constructor accepts the tkinter root window, sets a title, prepares helper objects (ModelInfo for descriptions and OOPExplainer for instructional text), initializes state (current\_model and input\_data), and calls setup\_ui to lay out the interface. This clean separation of concerns—initialization versus layout—improves readability and maintainability.



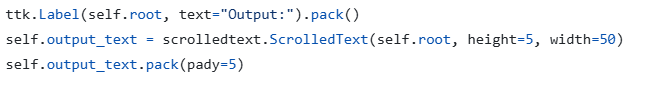
The first segment of setup\_ui constructs the input controls: a label, a combobox to choose between text and image inputs, and a “Load File” button. The combobox is bound to on\_type\_change, which resets any previously loaded data when the user switches modalities. This interaction pattern prevents mismatched inputs (e.g., an image path being treated as text).



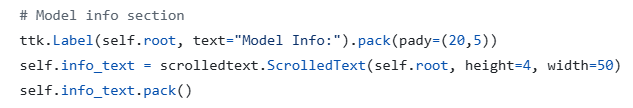
The next block sets up model selection using radio buttons backed by a StringVar. Each radio button chooses a specific model type and triggers on\_model\_change, which will re-instantiate the AIModel wrapper and refresh the model information panel accordingly. The default selection is sentiment analysis, ensuring the UI is immediately functional on startup.



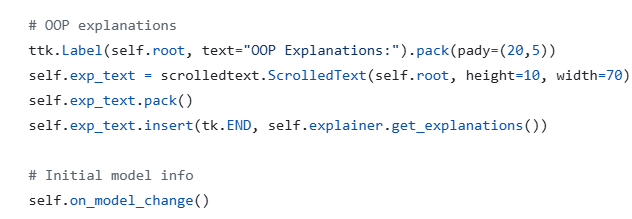
This button invokes the core action: running the selected model on the loaded input. It points to \_decorated\_run\_model, which is wrapped by the earlier decorator to log button clicks. Centralizing execution in one method simplifies validation and error handling.



There is an Output pane—a scrollable text box that shows whatever the model returns. When you run sentiment analysis or image classification, it prints the predicted label; for text-to-image, it confirms that the picture was created and saved. Making this area scrollable keeps long messages manageable now and leaves room for richer output in the future.



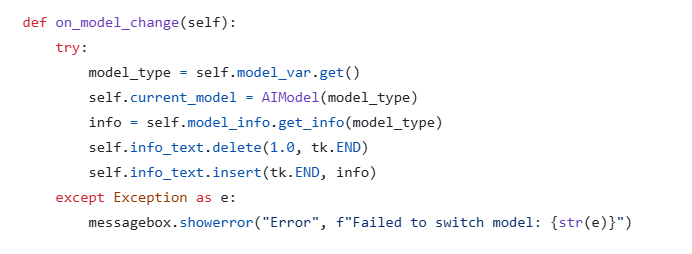
This section presents a human-readable description for the currently selected model. It is populated from ModelInfo via on\_model\_change. Providing inline guidance improves usability and pedagogical clarity, especially for users who are new to these model families.



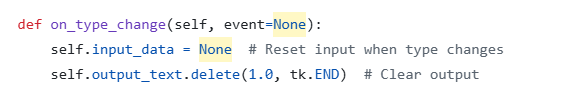
Finally, there is a small panel in the window that displays short notes from OOPExplainer, underscoring that the app is meant to teach as well as run models. During setup, the program calls on\_model\_change() so the default model’s details are already filled in when the window opens, which makes the interface feel informative from the very first click.



This decorated handler orchestrates validation, loading, and prediction. It first verifies that data is loaded and that a model is selected, then enforces modality compatibility (image model requires image input; sentiment and text-to-image require text). Upon passing checks, it loads the model (downloading weights if needed), performs inference, and writes the result to the output area. Any failure is surfaced via a message box rather than a crash, maintaining a smooth user experience.



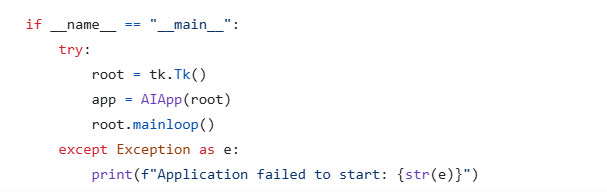
on\_model\_change synchronizes internal state with the user’s radio-button selection. It constructs a fresh AIModel wrapper for the chosen type and refreshes the “Model Info” panel from ModelInfo. Wrapping this flow in a try/except ensures that a failure to initialise a model does not destabilize the interface.



This simple handler resets previously loaded data and clears any displayed result whenever the user switches between text and image input modes. It prevents accidental reuse of an incompatible input and signals clearly that a fresh load is required.



load\_file prompts the user for a file appropriate to the selected input type and then loads it into memory: text files are read into a string; image files are represented by their paths. Errors are handled gracefully, distinguishing missing files from other exceptions and preserving the app’s responsiveness.



The entry point constructs the tkinter root window, instantiates AIApp, and starts the GUI event loop. The startup sequence is wrapped in a try/except so that any initialization failure produces a clear console message rather than an abrupt termination. This final block ensures the script runs as a standalone desktop application.

In summary, this Python application is more than a simple interface for AI models; it is a structured demonstration of how OOP principles can be embedded into real-world programming tasks. Encapsulation is evident in the controlled access to model information, inheritance provides a consistent framework for multiple model classes, and polymorphism allows the same interface to handle diverse input types. Multiple inheritance is used elegantly to add logging behavior, while decorators showcase the extensibility of Python functions. By combining these design elements with practical AI functionality and a GUI, the program provides both a working demonstration of machine learning tools and a rich educational example of object-oriented software engineering. It shows how clear class structures, reusable patterns, and robust error handling can make a complex task both accessible to end-users and pedagogically valuable to learners.